

注：本次调试步骤为以commit 417117eac3和7.29日最新的commit作为对比参照

复现 test_wint8() 单测

首先，复现 test_wint8() 这个单测错误的命令为

```
# current's dir is PaddleNLP/
python -m pytest -s -v
tests/l1m/test_predictor.py::BlockAttnPredictorTest_0__internal_testing__tiny_
fused_llama_inference5_2::test_wint8 > test_check.log 2>&1

# export test_wint8() model
python export_model.py --model_name_or_path /root/autodl-
tmp/models/models/__internal_testing__/tiny-fused-llama-inference5.2 --
output_path /root/autodl-tmp/models/exported_model/tiny-fused-llama-inference5.2
--dtype float16 --inference_model --block_attn 1 --quant_type 1
```

由宏观到微观的精度对齐

Step 1 逐层打印并检查

最开始需要从宏观的角度来查看每一层的output tensor的值，可以在 predictor.run() 之前加入 hook_func

```
def hook_func(op_type: str, tensor_name: str, tensor: paddle.Tensor):
    # ... 在这里可以求和、求均值、求方差或者直接输出全部数据等等
    print(op_type, tensor_name, tensor)
    # ...

predictor.register_output_hook(hook_func)
```

这里，是需要给 staticBlockInferencePredictor 这个类中 _create_predictor 这个函数最后一行加代码如下

```
def hook_func(op_type: str, tensor_name: str, tensor: paddle.Tensor):
    print(op_type, tensor_name, tensor)
self.predictor.register_output_hook(hook_func)
```

这里可以找出第一次出错的tensor，由于我最开始认定block_attn有精度问题，因此，主要检查这个算子的输出，最终在第10个block_attention发现出错。

Step 2 每次block_attention算子中打印中间变量

通过VLOG和VLOGMatrix函数进行打印，主要打印的位置及代码如下

```
/*
*
path:/home/Paddle/paddle/phi/kernels/fusion/gpu/block_multi_head_attention_kerne
1.cu
* 刚进入DispatchwithDtype函数时
*/
```

```

        DenseTensor* value_cache_out) {
    phi::DenseTensor qkv_buf;
    phi::DenseTensor fmha_buf;
    VLOGMatrix(qkv.data<T>(), 10, "***qkv pos1***", 10);
    VLOG(1) << "fmha_out " << fmha_out->dims();
    if (out_scale <= 0) {
        dev_ctx.template Alloc<T>(fmha_out);
        fmha_buf = *fmha_out;
    } else {
        fmha_buf.Resize(fmha_out->dims());
        dev_ctx.template Alloc<T>(&fmha_buf);
        dev_ctx.template Alloc<int8_t>(fmha_out);
    }
    VLOGMatrix(fmha_buf.data<T>(), 10, "***fmha_buf pos1***", 10);

    // InitValue(dev_ctx, fmha_buf.data<T>(), fmha_buf.numel(), static_cast<T>(0.));
    VLOGMatrix(fmha_buf.data<T>(), 10, "***fmha_buf pos11***", 10);
    const auto& input_dims = qkv.dims();

/*
* 退出DispatchWithDtype函数之前
*/
    if ((*qkv_out).initialized()) {
        VLOGMatrix((*qkv_out).data<T>(), 10, "qkv_out_buf final", 10);
    } else {
        LOG(INFO) << "qkv_out is nullptr";
    }

    if ((*fmha_out).initialized()) {
        VLOGMatrix((*fmha_out).data<T>(), 10, "fmha_out_buf final", 10);
    } else {
        LOG(INFO) << "fmha_out is nullptr";
    }

    if ((*key_cache_out).initialized()) {
        VLOGMatrix((*key_cache_out).data<T>(), 10, "key_cache_out final", 10);
    } else {
        LOG(INFO) << "key_cache_out is nullptr";
    }

    if ((*value_cache_out).initialized()) {
        VLOGMatrix((*value_cache_out).data<T>(), 10, "value_cache_out final", 10);
    } else {
        VLOG(3) << "value_cache_out is nullptr";
    }
}

```

Step 3 根据打印log分析出错位置

(需要根据导出的pdmodel来追溯出现错误的起因)

在打印了一些变量的值后可以发现，在第10轮计算block_attn的时候，输入的qkv就已经开始有误差了。

变量打印的位置如下：

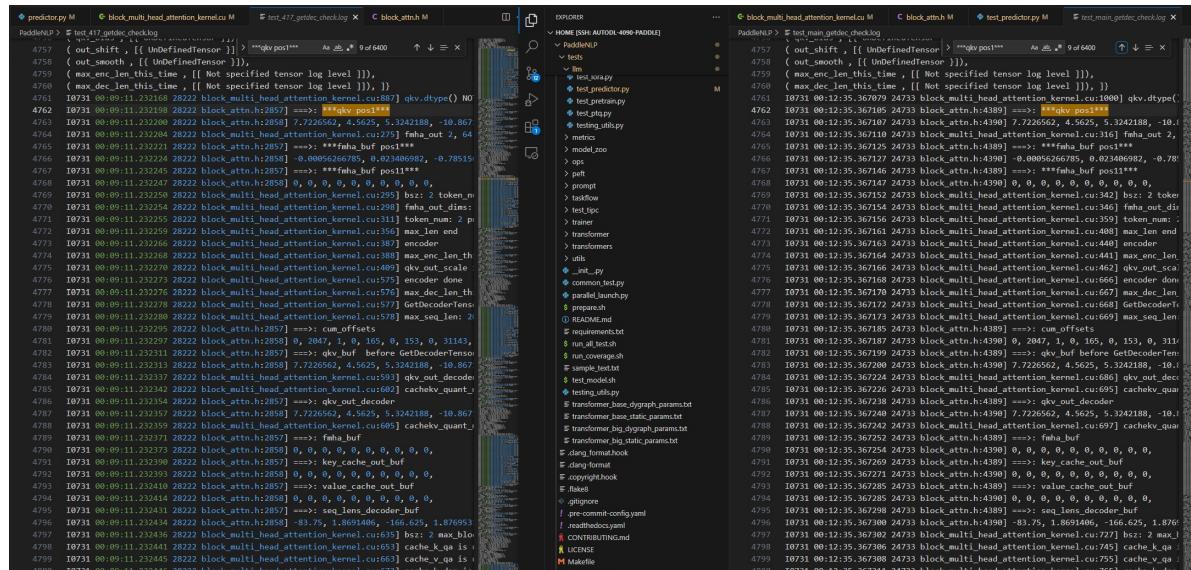
```

    const float quant_max_bound,
    const float quant_min_bound,
    const float out_scale,
    const std::string& compute_dtype,
    DenseTensor* fmha_out,
    DenseTensor* qkv_out,
    DenseTensor* key_cache_out,
    DenseTensor* value_cache_out) {
    phi::DenseTensor qkv_buf;
    phi::DenseTensor fmha_buf;
    VLOGMatrix(qkv.data<T>(), 10, "***qkv pos1***", 10);
    VLOG(1) << "fmha_out " << fmha_out->dims();
    if (out_scale <= 0) {
        dev_ctx.template Alloc<T>(fmha_out);
        fmha_buf = *fmha_out;
    } else {
        fmha_out.Resize(fmha_out->dims());
        dev_ctx.template Alloc<T>(&fmha_buf);
        dev_ctx.template Alloc<int8_t>(fmha_out);
    }
    VLOGMatrix(fmha_buf.data<T>(), 10, "***fmha_buf pos1***", 10);

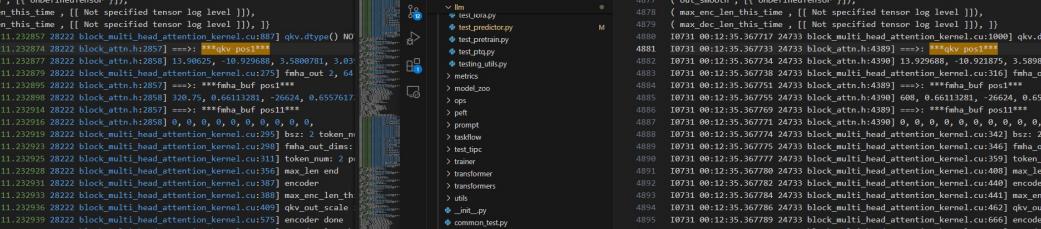
    InitValue(dev_ctx, fmha_buf.data<T>(), fmha_buf.numel(), static_cast<T>(0.));
    VLOGMatrix(fmha_buf.data<T>(), 10, "***fmha_buf pos11***", 10);
    const auto& input_dims = qkv.dims();
}

```

第九轮一些变量的值是这样的，可以发现所有tensor的值都是一样的



第10轮，可以发现已经有一些tensor的值出现了微小的精度差异，追根溯源，发现在一进来的时候qkv的值就不一样了。

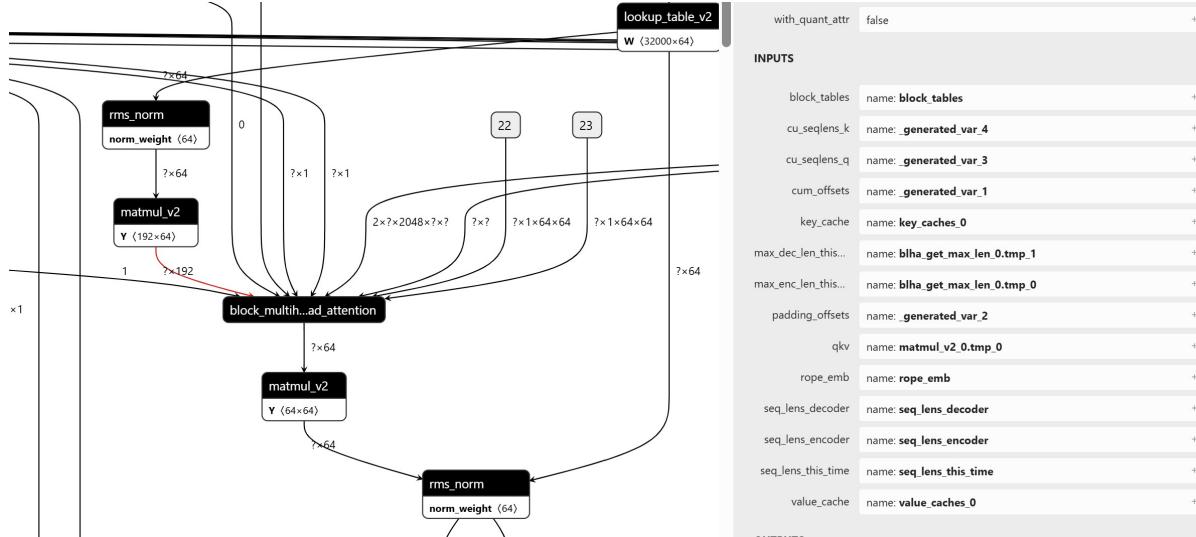


The screenshot shows a Windows Task Manager window with several tabs open, each representing a different GPU process. The tabs include:

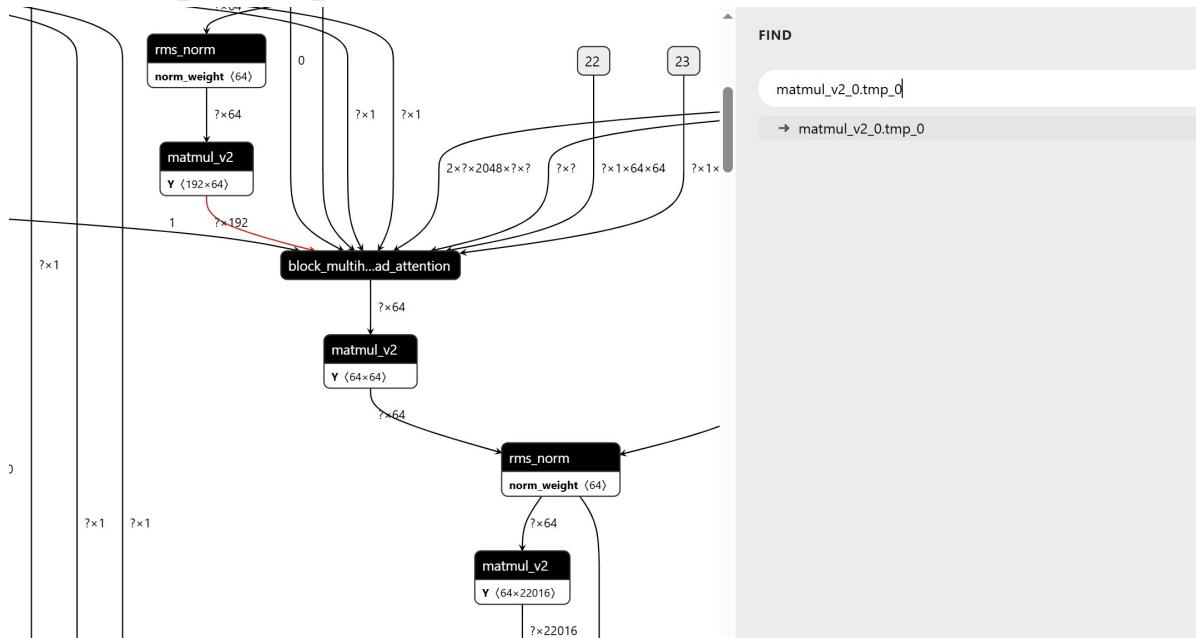
- predictor.py M
- block_multi_head_attention_kernel.cu M
- test_417_getter_check.log
- C block_attn.h
- ... C block_multi_head_attention_kernel.cu M
- C block_attn.h M
- D test_predictor.py M
- D test_main_getter_check.log

Each tab displays a list of GPU processes, likely CUDA threads or kernel invocations, with columns for PID, Process Name, and various performance metrics like CPU usage, memory usage, and GPU utilization.

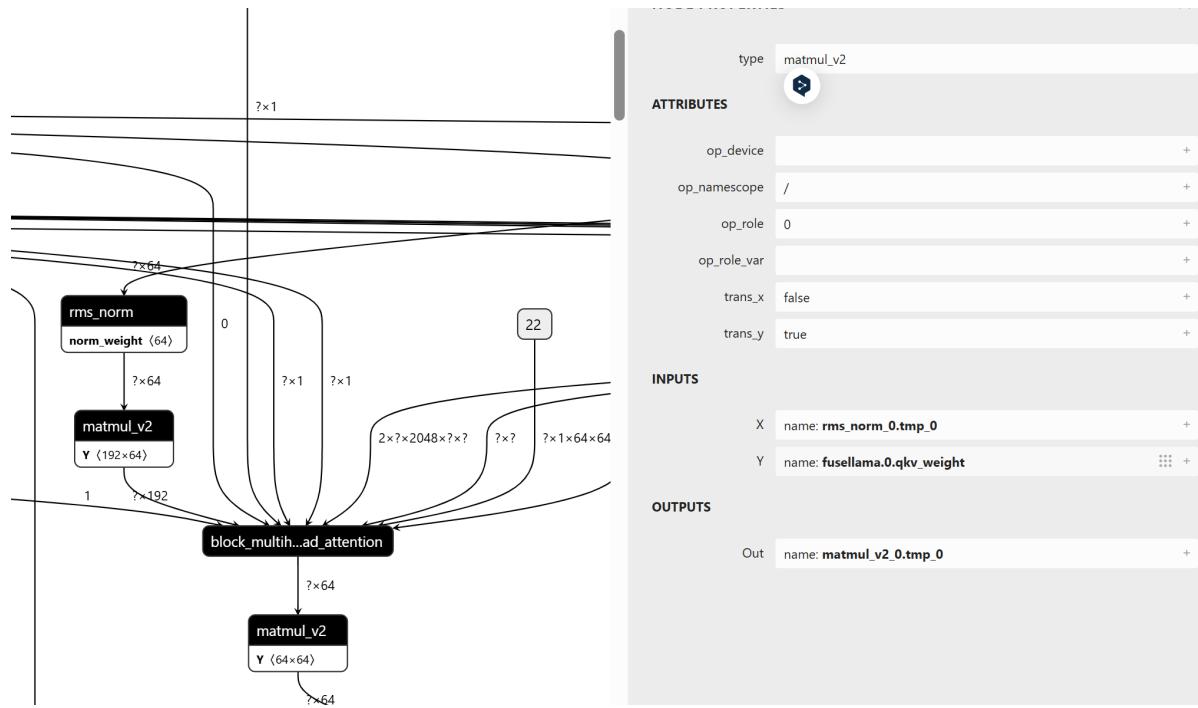
然后去netron中查找，可以发现这个qkv在图中名为matmul_v2_0.tmp.0



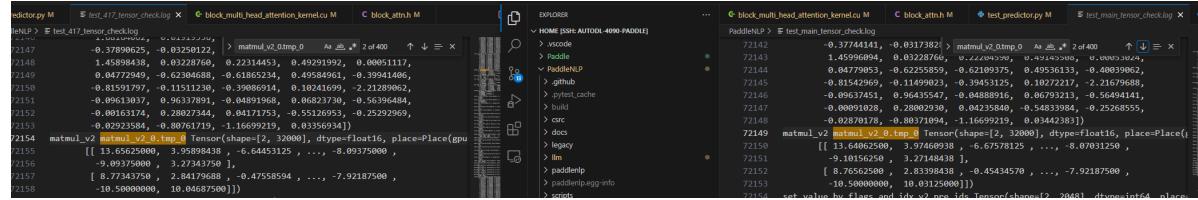
它来自于block_multihead_attention的上一个matmul。



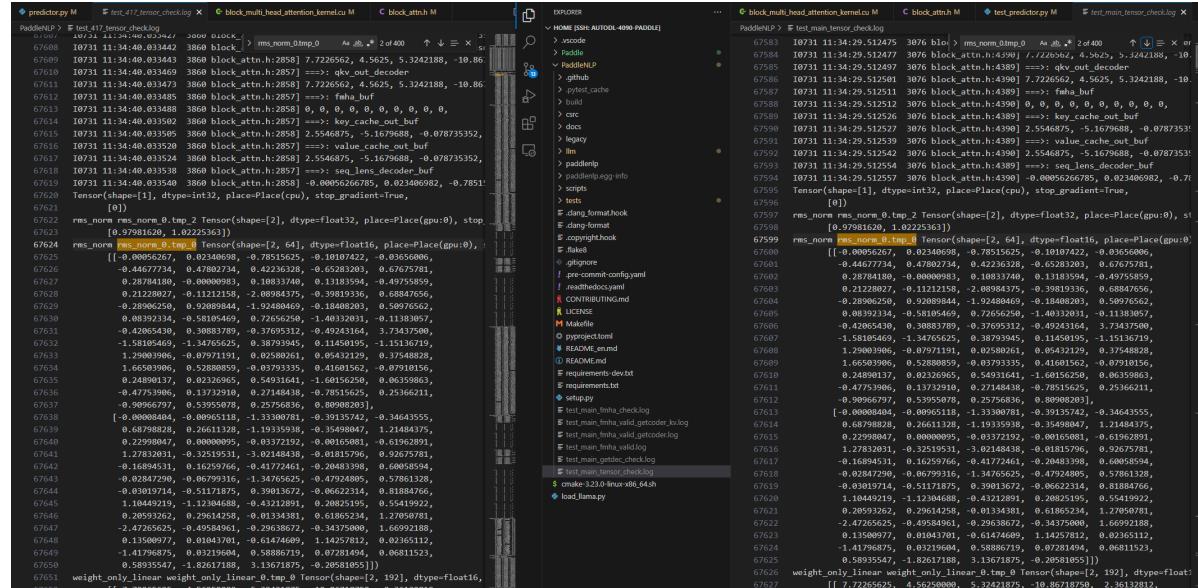
这里weight假设是肯定没有精度差异的。



去打印出来的tensor看，发现第二次计算得到的matmul_v2_0.tmp_0精度就不一样了。



因此继续去看到matmul_v2的输入rms_norm_0.tmp_0



```

diff --git a/test_wint8/test_main_tensor_check.log b/develop/test_main_tensor_check.log
--- a/test_wint8/test_main_tensor_check.log
+++ b/develop/test_main_tensor_check.log
@@ -1,10 +1,10 @@
 0.08856081, 1.48039862, 0.3248633, -0.16821289, 1.10253906, 0.34844453,
-0.7026672, 1.17382812, 0.59375000, -0.52148438])
 724 bilha_get_max_len bilha_get_max_len_0.tmp_1 Tensor(shape=[1], dtype=int32, place=Place{[75]}) 724 bilha_get_max_len bilha_get_max_len_0.tmp_1 Tensor(shape=[1], dtype=int32, place=Place{[75]}) 725 bilha_get_max_len bilha_get_max_len_0.tmp_0 Tensor(shape=[2], dtype=float32, places=Place{gpu:0}, stop_gradient=True)
 726 rms_norm rms_norm_0.tmp_2 Tensor(shape=[2], dtype=float32, places=Place{gpu:0}, stop_gradient=True)
 727 [ 0.9356801, 1.08763468]
 728 rms_norm rms_norm_0.tmp_2 Tensor(shape=[2], dtype=float32, place=Place{gpu:0}), [
 729 [ { 0.0049240, -0.01899719, -0.35498847, 0.11993408, -0.58066406,
 730 -0.08996582, 0.13422852, 0.77343750, 2.71093750,
 731 -0.09014893, 0.00801770, 0.08026123, 1.00878906, 0.04635268,
 732 0.06188965, 0.00031164, -0.00674824, -0.94824219, 0.68554688,
 733 -0.00486374, 0.24218750, 0.29174805, 0.4074870, 0.87548828,
 734 -0.24157715, 0.31056641, 3.4824188, -2.11523438, -0.92500916,
 735 0.11358643, 0.21545410, -0.66699219, 0.04211426, -0.52246694,
 736 1.46289862, 0.67488469, -0.52294922, 0.03244019, -0.83642578,
 737 0.2194696, 0.1000000, -0.0000000, 0.0000000, 0.0000000,
 738 0.05507423, 0.346653172, -0.1507402, 0.22530859, 0.00009989,
 739 0.22534319, 0.426699153, 1.44433594, 0.50244141, 0.03756244,
 740 -0.87841797, 0.17749023, -0.2956543, 0.41357427, -1.21269862,
 741 1.46386719, 1.97812590, 0.00547791, 1.25488281, -0.74072266,
 742 [-0.00907898, -0.02233887, -0.42163086, -0.16748847, 0.37953657,
 743 -0.22656250, 0.06042488, -0.9462896, -0.74072266,
 744 -0.15563965, 0.000000167, -0.22894727, 1.87402344, 0.58390625,
 745 1.02246994, 0.30737395, -2.23046875, -0.38696289, -1.38671875,
 746 0.08795166, 1.46484375, -2.25781250, -0.03677368, -0.59966938,
 747 0.0200000, 0.3979000, 0.0000000, 0.0000000, 0.0000000,
 748 -0.40389567, 0.13000768, -0.15270809, -0.04716777,
 749 -0.37189375, 0.44091797, 1.31047552, 1.31087427, -0.19409180,
 750 -1.13671875, 1.85546875, -0.06252586, 0.38964844,
 751 -1.54492188, -0.33911133, 0.269262711, -0.28710938, 0.03652344,
 752 -0.01386261, 0.01038975, 0.46484375, 2.21093750, 0.00167942,
 753 0.94628906, 0.03839111, 0.10888672, 0.86816406, -0.54882812,
 754 -0.3928227, 1.53027344, -0.98673828, 0.64453125)])
 755 weight_only_linear weight_only_linear_0.tmp_0 10731 11:34:40.1838847 3860 block_attn.h
 756 10731 11:34:40.183861 3860 block_attn.h:[2858] 0.23046875, -1.6298828, -8.65625, -8

```

可以发现，在所有数据中的第2和第3个rms_norm_0.tmp_0均无精度问题，（可以通过看行数发现第3个rms_norm_0.tmp_0已经在第二次计算得到的matmul_v2_0.tmp_0之后了）因此，目前推断matmul_v2有精度误差。而根据对比commit id 417...的逻辑，develop分支最新的commit下test_wint8单测的精度问题不在block_attn。

每个case fail和pass的情况

通过对单个不同input逐个分析发现，两个commit之前的对比如下（左边为417...,右边为develop最新commit）

这里batch size为2，也就是说20个test case，每个case跑两个input，实际跑的时候，共20个句子，每个句子有两次作为输入，第1-10次推理，每次两个句子，共跑10次，第11-20次推理，同上。

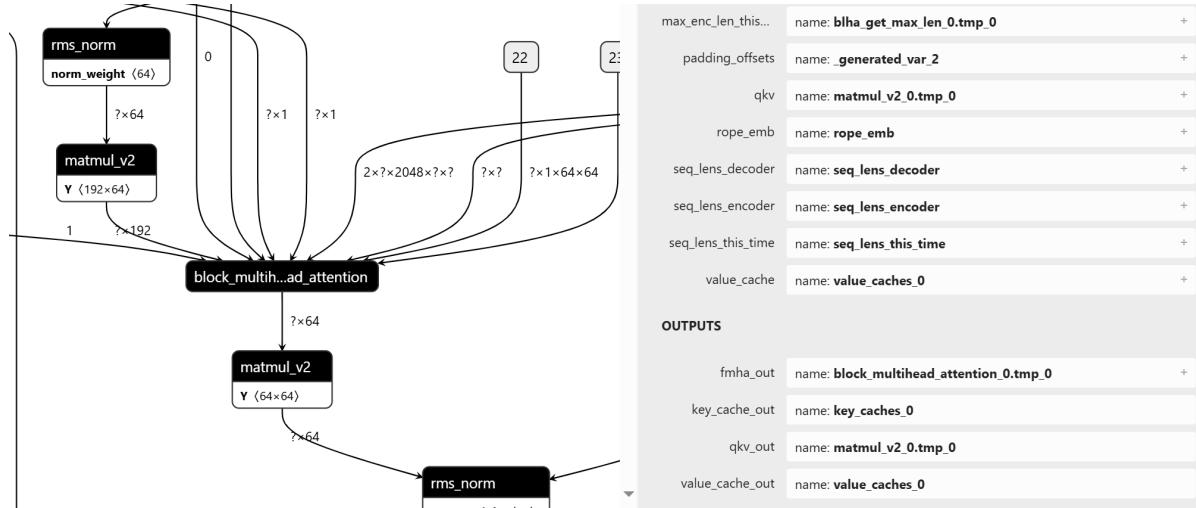
实际，判断的时候，用另一个命令把block_attn设置false，再把两个命令输出的句子进行对比。

179873	Twe ^紀 Intel междунаро ^д	179847	*****Output*****
179874	***NO 1 PASS***	179848	Twe ^紀 Intel международ ^紀
179875		179849	***NO 1 FAIL***
179876	***NO 2 PASS***	179850	
179877		179851	***NO 2 PASS***
179878	***NO 3 PASS***	179852	
179879		179853	***NO 3 FAIL***
179880	***NO 4 PASS***	179854	
179881		179855	***NO 4 FAIL***
179882	***NO 5 PASS***	179856	
179883		179857	***NO 5 PASS***
179884	***NO 6 PASS***	179858	
179885		179859	***NO 6 FAIL***
179886	***NO 7 PASS***	179860	
179887		179861	***NO 7 FAIL***
179888	***NO 8 PASS***	179862	
179889		179863	***NO 8 PASS***
179890	***NO 9 FAIL***	179864	
179891		179865	***NO 9 FAIL***
179892	***NO 10 PASS***	179866	
179893		179867	***NO 10 PASS***
179894	***NO 11 PASS***	179868	
179895		179869	***NO 11 FAIL***
179896	***NO 12 PASS***	179870	
179897		179871	***NO 12 FAIL***
179898	***NO 13 FAIL***	179872	
179899		179873	***NO 13 FAIL***
179900	***NO 14 PASS***	179874	
179901		179875	***NO 14 PASS***
179902	***NO 15 FAIL***	179876	
179903		179877	***NO 15 PASS***
179904	***NO 16 PASS***	179878	
179905		179879	***NO 16 PASS***
179906	***NO 17 PASS***	179880	
179907		179881	***NO 17 FAIL***
179908	***NO 18 PASS***	179882	
179909		179883	***NO 18 PASS***
179910	***NO 19 PASS***	179884	
179911		179885	***NO 19 PASS***
179912	***NO 20 PASS***	179886	
179913		179887	***NO 20 PASS***
179914	**len** : 20	179888	
179915	***full_match: ***: 1	179889	**len** : 20
179916	.	179890	***count*** 16
179917		179891	***full_match: ***: 10

左边过17个，右边过10个，根据之前的分析（用417...的第1个case比对最新commit的第一个case），发现精度的问题从matmul开始对不齐，但是经过后续分析可以发现，即使是用417...的第2个case比对最新commit的第2个case，也就是两个commit都能过的case，在后续也会出现同样的精度无法完全对齐的问题，可是单测都过了，也就是说在两边都能过的case中，我理解最终生成的token_id受这些精度问题的影响不是特别大。

根据上述想法，再打印一些更多的tensor数据出来，来判断上述想法的可能性。

首先，看看block_attn的最终输出在这些case中的区别，根据下图，block_attn的输出有四个，都打印出来。



第1个case (417-id, PASS, new-id, FAIL) NO1&NO2

```
  predictor.py M   test_predictor.py M   block_multi_head_attention_kernel.cu M   test_case_checklog x  test_no_mvhv.log   block_3 ...   block_multi_head_attention_kernel.cu M   test_case_checklog x  test_result_check_log   C block_attch.h M   test_predictor_z.M

Ruddington@E:\test_case_checklog> ./block_multi_head_attention_kernel.cu M
Ruddington@E:\test_case_checklog> ./test_no_mvhv.log
Ruddington@E:\test_case_checklog> ./block_3 ...
Ruddington@E:\test_case_checklog> ./block_multi_head_attention_kernel.cu M
Ruddington@E:\test_case_checklog> ./test_case_checklog x
Ruddington@E:\test_case_checklog> ./test_result_check_log
Ruddington@E:\test_case_checklog> ./C block_attch.h M
Ruddington@E:\test_case_checklog> ./test_predictor_z.M
```

和之前的分析一样，最开始的几个轮次精度正常，从第10轮开始出现精度问题。

看看该case下的最终输出，最终输出中key_cache和value_cache一样，qkv_out个别元素有较大误差，绝对误差在[0, 5]之间，fmha_out误差较大在[0,15]之间

这个case实际输出了两个output，第一个input pass，第二个input fail

第2个case (417-id, PASS, new-id, PASS) ALL PASS NO19&NO20

情况同第一个，虽然都该case两个commit都pass了，但是也存在精度误差。从第17轮开始出现精度问题

最终的输出，有误差，但不大，两个input都pass

情况一样的是，这里的误差，也来自于该次层input的qkv

检查最后一个case (最后两个可以pass的input) 引入误差的layer

第一个matmul_v2_0.tmp_0 (有误差)

第一个rms_norm_0.tmp_0 (matmul_v2_0.tmp_0的输入)

第二个rms_norm_0.tmp_0 (matmul_v2_0.tmp_0的输入)

```
predictor.py 3.M test_case_tensor_checklog test_case_predictor.py M block_multi_head_attention_kernel.cu M block_attn.h M EXPLORER ... f test_case_check.log c block_attach_2.M test_predictor.py 2.M predictor.py 3.M test_case_tensor_checklog
PaddleNetD> f test_case_tensor_checklog
107324 0.48049416, -0.1431165, 1.26660156, -d rms_norm_out_mp_0 Au ab_> 382 of 400
107325 0.99853516, 1.54687398, 0.2945738, 0.1037811, 0.62474671,
107326 0.1406973, 0.14471749, 0.11511234, 0.06171219, 0.048001367
107327 0.1406973, 0.14471749, 0.11511234, 0.06171219, 0.048001367
107328 0.82421875, 1.62267037, -0.6152383, -0.06573486, 0.38701172,
107329 0.2567656, -2.0312500, -0.95166016, -1.3687812, -0.81512344,
107330 0.3374627, -0.8453781, 0.5278330, -1.0438781, 0.6454313,
107331 0.1406973, 0.14471749, 0.11511234, 0.06171219, 0.048001367
107332 0.1053806, -0.40666707, 0.07795078, 0.4806820, 1.05664862,
107333 0.09775988, 0.14988646, -0.23791591, -0.77958781, -1.13885938,
107334 0.7211914, 0.21826172, 0.59578132, 0.07246967, 0.14487545,
107335 0.3542840, 0.37793928, -1.00353906, 0.18284244, 0.03911541,
107336 0.2799068, 0.49536013, 0.26446444, 1.1705332, 0.21020242,
107337 0.2799068, 0.49536013, 0.26446444, 1.1705332, 0.21020242,
107338 0.2799068, 0.49536013, 0.26446444, 1.1705332, 0.21020242,
107339 0.2799068, 0.49536013, 0.26446444, 1.1705332, 0.21020242,
107340 0.2799068, 0.49536013, 0.26446444, 1.1705332, 0.21020242,
107341 0.14257812, 0.12249456, 0.72134453, 0.71913591]]]
107342 bila_get_max_len_bila_get_max_len_mp_1 Tensor(shape=[1], dtype=int32, place=Place(cpu), stop_gradient=True)
107343 bila_get_max_len_bila_get_max_len_mp_1 Tensor(shape=[1], dtype=int32, place=Place(cpu), stop_gradient=True)
107344 bila_get_max_len_bila_get_max_len_mp_1 Tensor(shape=[1], dtype=int32, place=Place(cpu), stop_gradient=True)
107345 rms_norm_rms_norm_out_mp_2 Tensor(shape=[2], dtype=float32, place=Place(gpu)), stop_gradient=True,
107346 [0.10644725, 0.1406973, 0.14471749, 0.11511234, 0.06171219, 0.048001367]
107347 rms_norm_rms_norm_out_mp_2 Tensor(shape=[2], dtype=float16, place=Place(gpu)), stop_gradient=True,
107348 [0.10644725, 0.1406973, 0.14471749, 0.11511234, 0.06171219, 0.048001367]
107349 [-0.10180259, 0.08076849, 0.63233242, 0.17684668, -0.88464756, 1.12403344,
107350 0.77671788, 0.0004178, 0.22144535, 0.1818477, 0.47353732, 0.10538062,
107351 0.10180259, 0.08076849, 0.63233242, 0.17684668, -0.88464756, 1.12403344,
107352 0.08041187, 0.83762617, 0.4275125, 0.75800008, -0.99589556,
107353 0.40682242, 0.24707018, -0.31426898, 0.05545844, -0.47875977,
107354 -0.0956188, 0.68839844, -3.18359375, -0.16162189, 0.24965680,
107355 0.6757188, 0.0004178, 0.22144535, 0.1818477, 0.47353732, 0.10538062,
107356 0.0625618, -0.81026154, -1.44726562, -0.35808308, 0.89979873,
107357 0.32226592, -0.18854199, 0.18676758, -0.36566406, -0.52409334,
107358 0.7234445, 2.29767002, -0.3197579, 0.81398438, 0.25254646,
107359 0.545313, 0.57145438, 0.55581947, 0.87362087, 1.7885460,
107360 0.5651979, 0.80080257, 0.11669923, 1.62869025, 0.58034766,
107361 0.44155839, 0.22095117, 0.74416982, 1.60433194, 0.98081768,
107362 0.04773931, 0.9354604, -0.97191141, 0.8998484, 0.23293597,
107363 0.5774443, 0.23474121, 0.11573669, -0.21145151, 0.16103375,
107364 0.5774443, 0.23474121, 0.11573669, -0.21145151, 0.16103375,
107365 0.13738281, 0.30880203, 0.98437598, -0.01612465, -1.33307871,
107366 0.34138059, 0.83544922, 0.04742432, -0.07885742, 0.84899176,
107367 0.0324696, 2.80654688, -0.10485846, 0.07080732, 0.24607773,
107368 0.74121894, 0.83360141, 0.74462893, 0.13671881, 0.39211887,
107369 -0.74121894, 0.83360141, 0.74462893, 0.13671881, 0.39211887,
107370 0.10156258, 0.16735340, -1.15288212, 0.49378474]]]
107371 weight_only_linear_weight_only_linear_p_0 1073171516.733808 30846 block_attn_h:285] >>> :***gkv
107372 0.298573, 0.30846 block_attn_h:285]]]
107373 0.298573, 0.30846 block_attn_h:285]]]
107374 0.298573, 0.30846 block_attn_h:285]]]
107375 0.298573, 0.30846 block_attn_h:285]]]
107376 0.298573, 0.30846 block_attn_h:285]]]
107377 0.298573, 0.30846 block_attn_h:285]]]
107378 0.298573, 0.30846 block_attn_h:285]]]
107379 0.298573, 0.30846 block_attn_h:285]]]
107380 0.298573, 0.30846 block_attn_h:285]]]
107381 0.298573, 0.30846 block_attn_h:285]]]
107382 0.298573, 0.30846 block_attn_h:285]]]
107383 0.298573, 0.30846 block_attn_h:285]]]
107384 0.298573, 0.30846 block_attn_h:285]]]
107385 0.298573, 0.30846 block_attn_h:285]]]
107386 0.298573, 0.30846 block_attn_h:285]]]
107387 0.298573, 0.30846 block_attn_h:285]]]
107388 0.298573, 0.30846 block_attn_h:285]]]
107389 0.298573, 0.30846 block_attn_h:285]]]
107390 0.298573, 0.30846 block_attn_h:285]]]
107391 0.298573, 0.30846 block_attn_h:285]]]
107392 0.298573, 0.30846 block_attn_h:285]]]
107393 0.298573, 0.30846 block_attn_h:285]]]
107394 0.298573, 0.30846 block_attn_h:285]]]
107395 0.298573, 0.30846 block_attn_h:285]]]
107396 0.298573, 0.30846 block_attn_h:285]]]
107397 0.298573, 0.30846 block_attn_h:285]]]
107398 0.298573, 0.30846 block_attn_h:285]]]
107399 0.298573, 0.30846 block_attn_h:285]]]
107400 0.298573, 0.30846 block_attn_h:285]]]
107401 0.298573, 0.30846 block_attn_h:285]]]
107402 0.298573, 0.30846 block_attn_h:285]]]
107403 0.298573, 0.30846 block_attn_h:285]]]
107404 0.298573, 0.30846 block_attn_h:285]]]
107405 0.298573, 0.30846 block_attn_h:285]]]
107406 0.298573, 0.30846 block_attn_h:285]]]
107407 0.298573, 0.30846 block_attn_h:285]]]
107408 0.298573, 0.30846 block_attn_h:285]]]
107409 0.298573, 0.30846 block_attn_h:285]]]
107410 0.298573, 0.30846 block_attn_h:285]]]
107411 0.298573, 0.30846 block_attn_h:285]]]
107412 0.298573, 0.30846 block_attn_h:285]]]
107413 0.298573, 0.30846 block_attn_h:285]]]
107414 0.298573, 0.30846 block_attn_h:285]]]
107415 0.298573, 0.30846 block_attn_h:285]]]
107416 0.298573, 0.30846 block_attn_h:285]]]
107417 0.298573, 0.30846 block_attn_h:285]]]
107418 0.298573, 0.30846 block_attn_h:285]]]
107419 0.298573, 0.30846 block_attn_h:285]]]
107420 0.298573, 0.30846 block_attn_h:285]]]
107421 0.298573, 0.30846 block_attn_h:285]]]
107422 0.298573, 0.30846 block_attn_h:285]]]
107423 0.298573, 0.30846 block_attn_h:285]]]
107424 0.298573, 0.30846 block_attn_h:285]]]
107425 0.298573, 0.30846 block_attn_h:285]]]
107426 0.298573, 0.30846 block_attn_h:285]]]
107427 0.298573, 0.30846 block_attn_h:285]]]
107428 0.298573, 0.30846 block_attn_h:285]]]
107429 0.298573, 0.30846 block_attn_h:285]]]
```

打印test case

如果需要具体查看那个test_case pass或者fail了可以稍微修改一下 test_wint8 的代码如下

```
def test_wint8(self):
    self.run_predictor({"inference_model": True, "quant_type": "weight_only_int8", "block_attn": True})
    result_0 = self._read_result(os.path.join(self.output_dir, "predict.json"))
    self.run_predictor({"inference_model": True, "quant_type": "weight_only_int8"})
    result_1 = self._read_result(os.path.join(self.output_dir, "predict.json"))

    assert len(result_0) == len(result_1)
    count, full_match = 0, 0
    i = 0
    for inference_item, no_inference_item in zip(result_0, result_1):
        min_length = min(len(inference_item), len(no_inference_item))
        count += int(inference_item[: min_length // 2] ==
no_inference_item[: min_length // 2])
        full_match += int(inference_item[:min_length] ==
no_inference_item[:min_length])
        i += 1
        if inference_item[:min_length] == no_inference_item[:min_length]:
            print("****NO %d PASS***\n%i")
        else:
            print("****NO %d FAIL***\n%i")

    print(" **len** :", len(result_0))
    print("****full_match: ***:", full_match)
    self.assertGreaterEqual(full_match / len(result_0), 0.75)

    if self.model_name_or_path == "__internal_testing__/tiny-fused-chatglm__":
        self.assertGreaterEqual(count / len(result_0), 0.3)
    else:
        self.assertGreaterEqual(count / len(result_0), 0.4)
```